Object-oriented methods in data engineering

Briales, M.J.; de Troyer, O.M.F.; Dijkstra, J.; Meersman, R.A.; Weigand, H.

Publication date:
1991

Document Version
Publisher's PDF, also known as Version of record

[redacted]

ITK RESEARCH REPORT

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE

ITK Research Report
May 21, 1991

Object-oriented methods in data engineering

M. Briales, O. De Troyer, J. Dijkstra,
R. Meersman, H. Weigand,

No. 26

SPRITE is an ESPRIT II project started in 1989 and developing a system for technical documentation. The partners are:

Océ Nederland (NL)
AEG Elektrokom (FRG)
TITN/Alcatel (F)
Trinity College Dublin (IR)
KUB/EIT (NL)

This report contains two articles written by members of the EIT group that is responsible for the design of the multimedia database:

Dijksta, J. De Troyer, O., Meersman, R., Weigand, H.: RIDL* as a software engineering aid - some practical results. In: Habrias, H. (ed.), Proc. 4th Filin Conference of methods and tools as aids to design information systems. Nantes, Sept. 1990.. (12p).

Briales, M.J., De Troyer, O.: Object-oriented integrity enforcement in a relational environment. Proc. 9th British National Conference on Databases. Wolverhampton, July, 1991. (31p).

ISSN 0924-7807

ESPRIT Project 2001

Storage
Processing and
Retrieval of
Information in a
Technical
Environment

Object-oriented methods in data engineering

RIDL* AS A SOFTWARE ENGINEERING AID

SOME PRACTICAL RESULTS

J.Dijkstra, O. De Troyer, R. Meersman, H. Weigand
Tilburg University

Abstract

This paper discusses the usage of the RIDL* tool in the design of information systems, and in particular the information server and multimedia database of the SPRITE system, a project within the European ESPRIT programme. The RIDL* tool is based on the NIAM (binary relationship model) methodology. It supports the development of the conceptual model, from the functional analysis up to the definition of the data structure. Particularly useful is the automatic generation of relational database schemas from the developed data model. We evaluate the usability of RIDL* and our detailed experiences with NIAM, as well as its compatibility with an object-oriented approach.

1. Introduction : ESPRIT project SPRITE

Storage **P**rocessing and **R**etrieval **I**n a **T**echnical **E**nvironment, SPRITE, is a project of the ESPRIT programme in which five European companies and universities are taking part. The goal of the SPRITE project is to develop a documentation system for technical environments, that is, an information management system in which it is possible to create and maintain documents by a sophisticated document processor or to extract information from other external resources, storing it afterwards in a multimedia database. SPRITE highly supports integration in existing environments, which is going to be realized by the paper entry and information acquisition components of the system. Since the documentation system should not only support text processing, but also the whole management of the documentation lifecyle, the system includes a document management and a browsing and retrieval function. The kernel of the system is the information server, which is the intermediary between the multimedia database and the applications.

The SPRITE system architecture is divided over three levels (Fig. 1). The central, or conceptual level, is the level of the information server. Here the SPRITE object types and methods are defined. On top of the information server, several applications are defined that use a uniform interface to the information server. The information server itself makes use of a physical level, comprised of a commercial relational database system and an optical disk server. Thus, although the SPRITE system is not an information system in the usual sense, it is built up in the same way. One may expect that the same methodology can be applied.

At the start of the project, the RIDL* tool (DeTroyer, Meersman & Verlinden, 1988; Intellibase, 1988; DeTroyer, 1989) has been chosen to support the design and implementation of the information server. RIDL* is based on NIAM (Nijssen, 1980; Verheijen & Bekkum, 1982). A short overview of RIDL* and its methodology is given in section 2. In section 3, we report on the actual outcomes of the design process. Finally, section 4 contains some evaluating comments, particularly on the compatibility of RIDL* with an object-oriented approach.

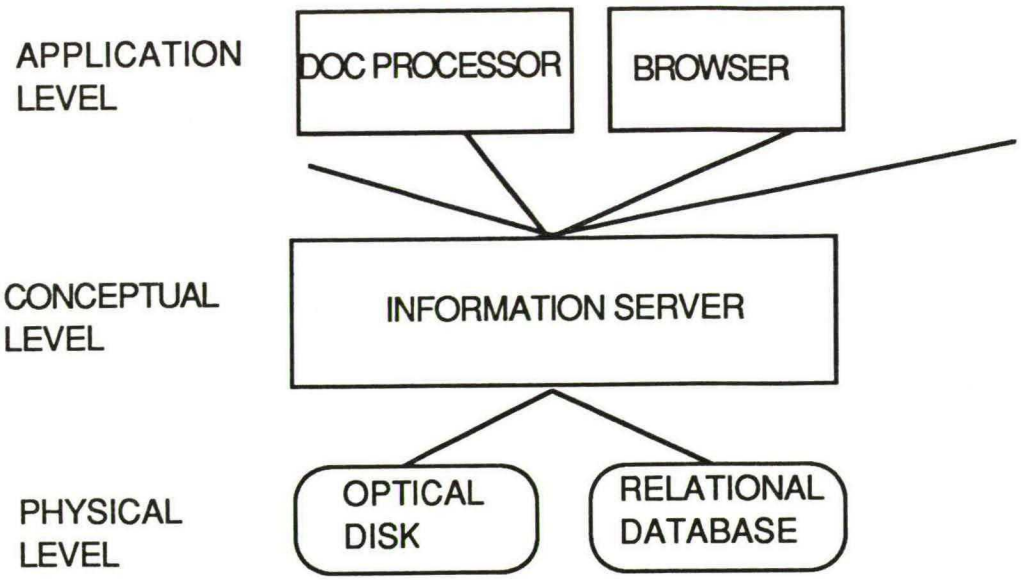

Fig 1 : SPRITE archtecture

2. Short overview of RIDL*

The RIDL* tool is a graphic-oriented design workbench consisting of several modules and supporting the design of information systems. It can be used profitably in the context of the NIAM design methodology (Nijssen, 1980; Verheijen & Bekkum, 1982), as described below.

The functional analysis phase, including the extraction and identification of the information requirements is supported by the RIDL-F module. Functional analysis in NIAM is done top-down: starting from the basic functions of the information system, the designer goes through an iterative process of functional decomposition until he encounters functions that (a) describe the transformation in full detail, and (b) for which the information flows can be defined precisely. With the use of RIDL-F, the designer can draw information flow diagrams (IFD) and decompose them on lower levels. This phase ends when an analysis and identification of the concepts in the system, extracted from the information flows at the lowest levels, become possible.

For the conceptual schema, NIAM uses a graphical notation that models the Universe of Discourse in the form of a semantic network. The model recognizes *object types* and associations (or *facts*). A distinction is made between lexical objects, such as "string", and non-lexical object types, such as "person". The facts can be annotated with cardinality constraints. It is possible to express subset relationships between object types (mutually exclusive or not, total or not).

RIDL-G supports the development of the conceptual schema by allowing the designer to draw NIAM pictures efficiently. Once the conceptual schema has been developed, it is very easy to maintain the schema as well thanks to a powerful graphical editor. When encountering non-graphical constraints affecting identified concepts or for additional comments, pop-up edit boxes enable the designer to document these where needed and at the time of developing the conceptual schema.

At each desired moment the user can inspect his schema against syntactic errors or other mistakes according to the rules of the binary relationship model. RIDL-A takes care of this. Before the RIDL-A module can access the conceptual schema, it must be stored in a RIDL

database, using RIDL-DBSTORE. After correct storage of the conceptual schema the RIDL-A module can be invoked in order to analyze the model.

The analysing part of the RIDL* tool consists of five phases :

naming analysis
completeness check
constraint validation
set contraint and consistency analysis
lexical referencibility analysis.

The naming analysis checks if the object types in the schema do have a unique name and checks naming-rules set to other concepts. The completeness analysis checks whether each fact between two object types does have arity and that subtypes of an object type do have a common supertype. The validation phase checks if the individual constraints set on objects are defined properly, while the validation of the combination of those constraints is checked in the set constraint consistency analysis. Finally the kernel of the RIDL-A module, the lexical referencibility analysis, checks whether each non-lexical object type is lexically referencible and generates for every non-lexical object type one or more lexical reference paths, to be used when mapping the conceptual schema into relational database schemas. For more details, see (DeTroyer, Meersman & Verlinden, 1988; Intellibase, 1988).

When the conceptual schema is considered to be syntactically correct the mapping module, RIDL-M, is invoked. This module takes care of the automatic generation of relational database schemas deduced from the developed conceptual schema. The relational database schemas are built in accordance with the specifications of a particular DBMS. Currently it is possible to generate relational database schemas for five different DBMS systems. Depending on the kind of DBMS, the mapper not only generates table definitions, but additional constraints and triggers as well.

The automatic generation of database schemas can be influenced. This is done by setting a number of mapping options. RIDL-M provides the storage of these mapping options and will use them when mapping the same conceptual schema, for example to see the impact of a slight modification in the mapping options on the database schemas.

Cross-references are made during the mapping and report the user afterwards how the conceptual schema was mapped to the generated database schemas (forward mapping) and how the generated database schemas correspond to the conceptual schema (backward mapping).

3. APPLYING RIDL* TO SPRITE

After this short overview of RIDL* and NIAM, we now go on to describe how this worked out in the SPRITE project. The reader should be aware of the fact that this description is from the perspective of our group, which is responsible for the information server (see Fig. 1). Therefore we deliberately ignore here the work on more peripheral application functions, such as scanning and picture recognition.

The first phase of the SPRITE project was reserved for analysis and design (about one year). Compared with the NIAM methodology described above, the project planning diverged on two points. First, it was deemed appropriate to generate prototype DB schemata in a very early stage; however, prototyping is not an "official" part of the NIAM methodology. Secondly, the top-down approach in functional decomposition could not be followed strictly, for two reasons. One reason had to do with the distributed and international character of the project,

which urged us to divide the work among modules as soon as possible. The functional decomposition requires that information flows between top level modules are fixed immediately. The other reason lies in the fact that the SPRITE system is not an information system in the usual sense. Therefore its functions are not so much information flows, as well as user tasks. We will come back to this in section 3.1.

The main divergence from "pure" NIAM was that, rougly speaking, we started the design of SPRITE with the conceptual model (middle layer - see Fig. 1), then the applications (top layer) and finally the physical level (bottom layer)

The global project planning consisted of the following steps (the numbers refer to the time sequence; the characters to different tasks within one stage):

(1) analysis and functional design total system
(2a) collection of information requirements
(2b) generation of prototype DB schemata
(3a) functional decomposition of the applications
(3b) functional specification information server
(4) conceptual model of database (structures and operations)
(5) integrated set of NIAM pictures/ update of prototype DB schemata

Stage (1) was a global effort of all partners. It identified the main functions of the system. The description was purely verbal, and no use of RIDL* was made yet.

In stage (2), each partner responsible for a certain application had to provide its information requirements, that is, its interface with the information server (conceptual level). RIDL-F was still not used, but RIDL-G was used for the data model part so that prototype DB schemata could be generated.

The functional analysis was done in stage (3). RIDL-F was used here as we will discuss shortly in more detail. In stage (4), the conceptual model was synthesized. The object classes could be identified and pictured in NIAM. Here RIDL-G proved to be of help (see below). In the last stage, the conceptual model was refined and adapted, and with the aid of RIDL-M, the final DB schemata could be generated.

Before focusing on the steps (3) and (4) in more detail, the following general remarks are due.

* the prototype DB schemata generated in stage (2) turned out to be less useful than expected. This was due to the fact that, since the functions were not defined yet, no real testing could be done already, and, secondly, the discussions at that time were still on the conceptual level.

* in contrast to NIAM, a strict sequencing of functional decomposition before object type specification was not aimed at. Right from the start of the design, it was felt necessary to discuss for example the definition of a document, its logical structure, versions of documents etc. Therefore NIAM pictures were already made for parts of the model long before the functional specification.

We now consider the functional specification and data structure design in more detail.

3.1. FUNCTIONAL DESIGN

The main task in the functional design phase is to identify what functions are relevant in the observed object system. In SPRITE, this was prepared in stage (1). On the basis of this, the applications could decompose the functions allotted to them, until information flows at information server level were identified. Simultaneously, the functions in the information server interface had to be defined. These two tasks are discussed here as *functional decomposition* and *functional specification.*

3.1.1. FUNCTIONAL DECOMPOSITION

The goal of the functional decomposition is to refine the basic functional requirements from stage (1) into well-specified modules and procedures. The functional decomposition was done for each application separately. Therefore a second goal, not less important, was to define the interfaces between the various applications.

RIDL-F could be used in this process profitably by providing a uniform and disiciplined format of specification. The decomposition was not continued to the bottom level, but as far as needed for a good overview of the information flows between different applications and between applications and information server.

As an example, fig 2 shows one level of the SPRITE system architecture, and in particular the desktop application. At the right of the big box, centered at the left of the figure, one can find the upper levels of the function box subject of decomposition (black boxes). As one can see, the main information flows were identified between the application interface (north-east in the IFD) and the information server (south-west).

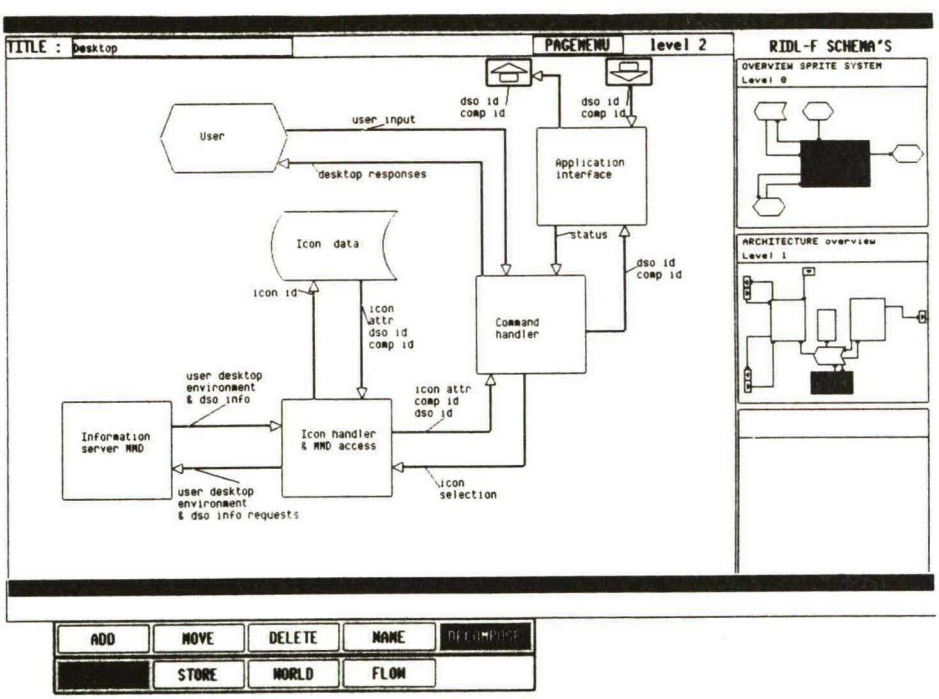

Fig 2 : Sample RIDL-F picture

A drawback was noted by the fact that RIDL-F does not separate control and information flow. One could also say that RIDL-F does support information flow only. An extension of RIDL-F in which control can be specified explicitly, is currently under development.

3.1.2. FUNCTIONAL SPECIFICATION

The goal of the functional specification was to specify the interface between applications and information server, or, put differently, to identify the functional part of the conceptual model. Note that for this goal, decomposition is not relevant, since an interface consists of one level. Instead, it is possible and useful to *categorize* functions.

The method we followed was based on Moran's task-oriented model CLG (Moran, 1981; 1983). In this model, a distinction is made between the task level, the semantic level, the syntactic level and the physical level. At task level, the tasks of the user are identified and decomposed if necessary on functional criteria. At the semantic level, elementary tasks correspond to operations, that may be complex. The syntactic and physical level describe how these operations are implemented. Note the difference between the "task" concept and the "function" concept used in the functional decomposition of NIAM where "function is the name we give to the capability to transform information flows" (Verheijen & Bekkum, p.541). In contrast, a task is defined in terms of the needs and goals of the user. Usually, the tasks descriptions do not have to do with information flow. For example, a task may be to create a document, or to edit a chapter. Tasks are defined in relation to objects rather than information, although the object may be an information object (like "document"). Therefore tasks, or, more precisely the semantic operations supporting the basic tasks, fit very well in the conceptual model we need for the specification of the information server.

One thing we did in the functional specification was to make a distinction between browsing and operating functions. Browsing functions only need retrieval functionality but don't cause updates in the database, whereas operating functions change the contents of the database. Browsing functions could be subcategorized further in navigation and retrieval. In this way, a systematic analysis of the functionality of the information server could be achieved that served as a basis for the conceptual model (objects and methods) developed later.
Although we used RIDL-F not in the way the methodology prescribes, the tool itself was useful in drawing pictures. However, RIDL-F does not recognize objects - functions operate on information flows, that may come from data stores, but objects, or object types, are not in the picture. A more object-oriented RIDL-F module, integrated with RIDL-G, would have been nice.

RIDL-F provides information flow diagrams only. A full specification of the operations, including its preconditions and postconditions, is not possible (yet?). Therefore no connection could be made with the next design step, the full conceptual model. For the specification of the dynamic part of the conceptual model, we defined methods for each object type. Methods were defined by means of preconditions, postconditions and triggers. The conditions were written in first-order logic with some syntactic sugar to improve readability (the style is based on the *language* RIDL as described in DeTroyer, Meersman & Ponsaert, 1983). Examples of methods are CREATE document, DELETE document, READ content, WRITE content, INSERT document INTO folder etc.

As a preliminary conclusion we can state that RIDL-F was instrumental as a tool for drawing function diagrams, but did not play the integrated role in the design process that it would have in the development of an information system along the principles of NIAM methodology.

3.2. DATA STRUCTURE DEFINITION

Above we discussed the functional design of the SPRITE system, now we focus on the data structures (objects).

One of the major and in fact one of the first steps in developing the conceptual model, is the

identification of the various object classes. Object classes are primarily consisting of attributes and relationships, which may be defined as attributes too, and methods, which define what operations can be performed on the object class.One of the reasons why NIAM may contribute in designing information systems following an object-oriented approach is that the NIAM methodology deals with object types very intensively.

NIAM makes a distinction between two kinds of object types: lexical and non-lexical, where the first kind can be considered as a lexical reference of the abstract meaning of a non-lexical object type. Lexical Object Types (LOT) are usually those concepts which can be expressed lexically, such as names, numbers, amounts, symbols , etc.. The opposite goes for NOn-Lexical Object Types (NOLOT). For instance, the name of a person is the lexical object type of the non lexical object type 'person'. Relationships (facts) exist between two object types and therefore validate the requirements of a binary relationship model like NIAM.

In the SPRITE project we felt the need to identify objects in an early stage. As a result the development of the datastructure and its graphical representation in NIAM was strongly emphasized. Therefore NIAM diagrams have been created early and were refined and adapted. Also during this period we felt we could use a kind of object-oriented approach. Class definitions could be deduced from the NIAM diagrams without great difficulties.

As an example, in the SPRITE project we have identified a number of spaces in which several objects occur. E.g. in the document space we have identified four (exclusive) document space objects : document, document list, version cluster and folder. Document space objects are always created at some time, can be modified, do always have a (non-unique) name and are always owned by some user (owner).

In NIAM terms, we got the diagram as in Fig 3. The diagram contains the constraints as well.

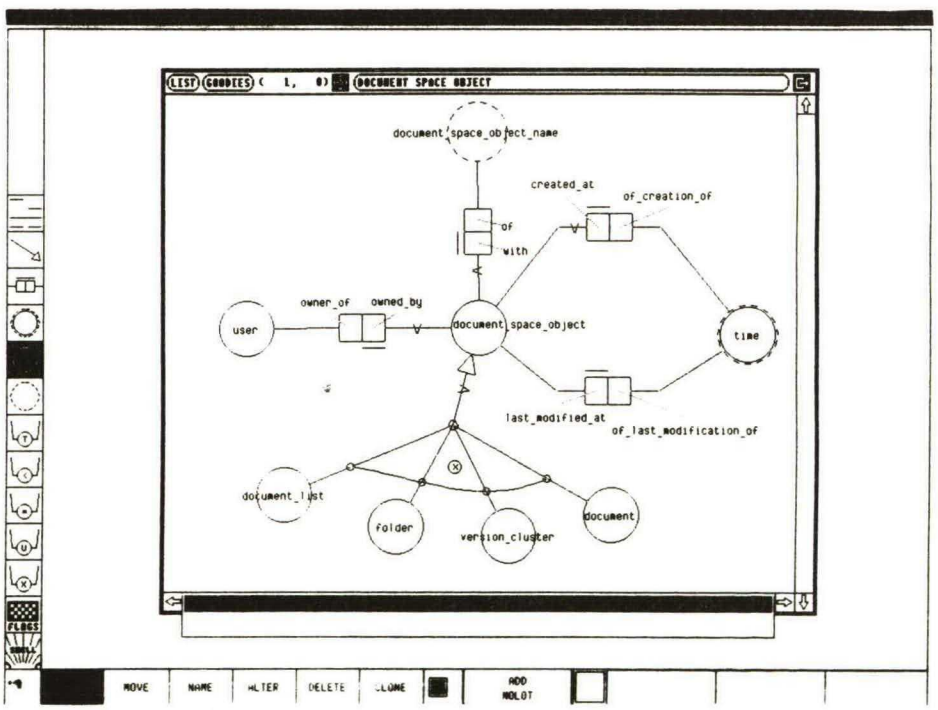

Fig. 3: sample RIDL-G picture

From this diagram we deduced the class definition of the document space object, and consists of four main parts : class description, attributes, subtypes and methods.

Class Document space object

An elementary document space object (document) or composed document space object (document-list, version cluster, folder)

Attributes

name: string;
created_at: time;
last_modified_at: time;
owned_by: user

Subtypes

{ document, folder, version cluster, document list }

Methods

CREATE
DELETE
INSERT
COPY
...

Note the similarity of the graphical representation in RIDL-G and the structure of the class definition.

RIDL-G allows the designer to define several constraints on the various concepts and some of these cannot be included in the class definitions. So, RIDL-G's contribution is of real importance to complete the model with graphical constraints. However, the non-graphical constraints must still be described 'by hand'. The domains of attributes (for example, "document space object name") are specified in the class definition, but not always in the RIDL-G picture.

When the project evolved, the graphical representation and the class definitions became closer related. Each one can be said to give a different "view" on the conceptual model, with its own merits and drawbacks.

Problems arose when object classes where defined such as an aggregation of object types, which is due to the fact that aggregation is not a concept in NIAM. We found a (partial) solution in the uniqueness constraint provided in NIAM, as we illustrate with the following example.

In the SPRITE system it was considered necessary to have versions of documents. In this example it is sufficient to know that a "configuration" is the identification of a document version. A configuration consists of a dimension and a particular value from the dimension's domain. More concrete : suppose we have a manual, and part of it describes keyboards of different kinds: VT100, VT220, etc.. Therefore the configuration dimension is 'keyboard' with configuration domain a set of configuration values {VT100, VT220}.

Taking an object-oriented stance, a configuration can be modeled as the aggregation of dimension and value, as follows:

Class configuration
Aggregation of
{configuration_dimension, configuration_value}

The only way to represent this in NIAM is by use of the uniqueness constraint (Fig.4). The objects included in the constraint (configuration dimension and configuration value) are uniquely identifying the object (configuration) on which the uniqueness constraint is defined. In other words, a configuration is exactly known when the configuration dimension with one corresponding value is known.

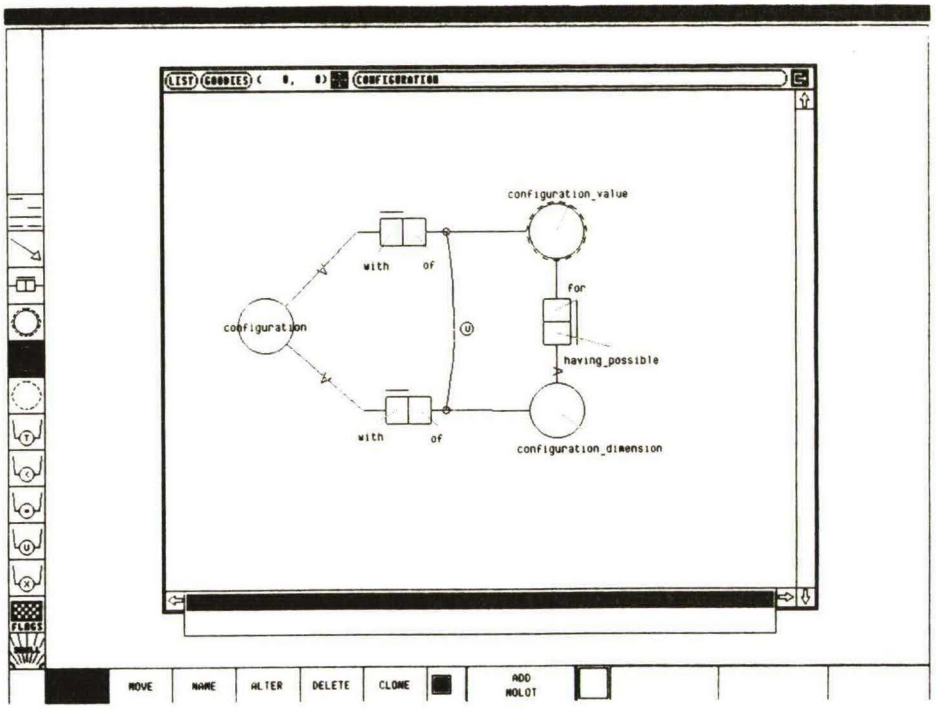

Fig. 4 : graphical representation of the aggregation concept in RIDL-G.

However, it might be argued that in this way we have lost the real meaning of aggregation and replaced it by a uniqueness constraint.

When defining object classes, relationships with internal structure were encountered as a problem in modelling according NIAM. For example, a composite component consists of several components in a particular sequence. If this ordering is important, as in the SPRITE project, we need to model it as well. But when we model only the relationship between composite component and component, -composite component contains component-, we have lost the semantics behind it. A simple solution was found in the object 'component occurrence' to be considered as an entry in the list of components making up the composite component. Now, the ordening has been modelled as a predecessor / successor -relationship defined on component occurtences. In this way, each component in a composite component is retrievable. As a consequence, the ordering was modelled without losing semantics but we had to introduce a new object.

The NIAM diagrams drawn by RIDL-G were not only useful in communicating with the project partners, but also served as input for the RIDL-A and RIDL-M modules. RIDL-A was used to check the consistency of the model, as described in section 2 above. RIDL-M was used to generate DB schemata. In the further progress of the project, additions and changes to the model could be processed easily and new DB schemata generated instantly.

In advance and during the mapping process mapping options can be set. They include for instance the mapping of sublink types, the control of admissibility of null values in tables, and the merging of keys in order to affect the size of a table in default - as well as for individual object cases. During the process data types can be assigned to the various object types.

In the example of the document space object, we have set an subtype indicator for the sublink mapping. That means that an additional column ("is_a_document" in Fig 5) has been added by RIDL-M to the table document space object, indicating whether a document is a document space object or not. (similar to the other subtypes of document space object). The admissibility of NULL values in columns was permitted for those columns which don't determine the primary key. Although it is not clear from the table itself whether a column is (part of) the primary key or not, this gap is bridged by the generated table constraints of a table, which indicate for instance what columns are making up the primary key and foreign key, what constraints are already implemented (unique index) and additionally what view constraints were computed on the table and still have to be implemented (Fig 5).

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                          TABLE doc_space_object                              */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

CREATE TABLE   doc_space_object
  ( object_id
      object_id                          /* DATA TYPE int */
      NOT NULL

  , doc_space_object_name_of
      doc_space_object_name              /* DATA TYPE 'varchar(30)' */
      NOT NULL

  , time_of_creation_of
      time                               /* DATA TYPE datetime */
      NOT NULL

  , name_owner_of
      name                               /* DATA TYPE 'varchar(50)' */
      NOT NULL

  , time_of_modification_of
      time                               /* DATA TYPE datetime */
      NULL

  , is_a_version_cluster
      b_is_a_1                           /* DATA TYPE tinyint */
      NULL

  , is_a_folder
      b_is_a_2                           /* DATA TYPE tinyint */
      NOT NULL

  , is_a_document_list
      b_is_a_3                           /* DATA TYPE tinyint */
      NOT NULL

  , is_a_doc
      b_is_a_4                           /* DATA TYPE tinyint */
      NOT NULL
  )
```

```
/*-------------------------------------------------------------------------------*
/*                Table Constraints For Table doc_space_object                   *
/*-------------------------------------------------------------------------------*
/*                                                                               */
/*sp_primarykey , object_id                                                      */
/*                                                                               */
/*sp_foreignkey  , object_id                                                     */
/*          object_reference , object_id                                         */
/*                                                                               */
/*sp_foreignkey  , name_owner_of                                                 */
/*          user , name                                                          */
/*-------------------------------------------------------------------------------*
/*                 View Constraints For Table doc_space_object                   *
/*-------------------------------------------------------------------------------*
/*                                                                               */
/*............................ :                                                */
.         ..............                                                        .
.         ..............                                                        .
/*                                                                               */
/*EXCLUSION VIEW CONSTRAINT :                                                    */
/*          ( SELECT object_id                                                   */
/*            FROM doc_space_object                                              */
/*            WHERE ( is_a_document_list = 1 )                                   */
/*          )                                                                    */
/*          IS EXCLUSIVE OF                                                      */
/*          ( SELECT object_id                                                   */
/*            FROM doc_space_object                                              */
/*            WHERE ( is_a_doc = 1 )                                             */
/*          )                                                                    */
/*          IS EXCLUSIVE OF                                                      */
/*          ( SELECT object_id                                                   */
/*            FROM doc_space_object                                              */
/*            WHERE ( checkpoint_tree_is_a is NOT NULL )                         */
/*          )                                                                    */
/*          IS EXCLUSIVE OF                                                      */
/*          ( SELECT object_id                                                   */
/*            FROM doc_space_object                                              */
/*            WHERE ( is_a_folder = 1 )                                          */
/*          )                                                                    */
/*-------------------------------------------------------------------------------*

/*-------------------------------------------------------------------------------*
/*             Implemented Constraints For Table doc_space_object                *
/*-------------------------------------------------------------------------------*

  CREATE UNIQUE INDEX  C_KEY_58 ON doc_space_object
   ( object_id )
```

Fig 5 : generated table (left) and table constraints (right).

At the moment of writing this paper, the generation of triggers and stored procedures (in SYBASE™; see Sybase, 1988) is under development. *Triggers* will prevent incorrect, unauthorized or inconsistent changes to the database and affect primarily the primairy key/foreign key integrity constraints, exclusion and subset constraints and the checking of rows. *Stored procedures* deal with primitive object manipulation and implement the insertion, updating and deletion of objects.

4. Methodological Issues

In this paper, we have described the design phase of SPRITE and how the RIDL* tool could be used profitably. The fact that SPRITE is not an information system required some adaptations in the methodology. The difference is that design starts with the conceptual model (beginning with the object types) rather than with information flows, as is commonly advocated in the literature.

Both RIDL-G and RIDL-F have proved useful in describing the conceptual model and information flows. The graphical notation was easy to learn for all project partners and facilitated the communication. Some problems with RIDL-F were detected, such as the expression of control information and the integration with RIDL-G.

The RIDL-A and RIDL-M modules were used to generate database tables. This saved us a lot of work, especially because later adaptations could be made directly on the conceptual level. In this way, the conceptual model and the database tables are always kept consistent.

RIDL-G has a high-level notation for object types and relationships. This module can be combined rather smoothly with what is called nowadays an object-oriented approach. From our own experience we tried to derive the rough outline of a design methodology based on these concepts as follows:

[1] IDENTIFYING OBJECTS AND TASKS.
In the first stage, the user's tasks are identified and the objects of these tasks. The object types can be modeled with RIDL-G. The tasks can be grouped according to the object types.

> *Example*: basic tasks are: EDIT document, PRINT document, INSERT chapter IN document, PUT document INTO folder, MAKE version of document, FIND all versions of document. Object types are : document, folder, chapter.

[2] FILLING IN THE CONCEPTUAL MODEL (STRUCTURES)
The basic tasks of stage [1] are decomposed to the semantic level. A difference is made between retrieval *functions* and *operations*. The first object types are refined (subtypes are distinguished if necessary) and augmented with secondary object types that pop up during the decomposition. Attributes and relations of object types are modeled.

> *Example*: the type "document" gets two subtypes, one for BuildingBlock documents and one for Composed Documents, to support the versioning mechanism. Chapter is generalized to "component". Operations on component are: CREATE, DELETE, COPY, INSERT, MOVE, MARK_CONFIGURATION ($i_conf) etc.

[3] DEFINING METHODS AND FUNCTION IMPLEMENTATIONS
The operations defined in [2] are specified for their preconditions, postconditions, and triggers (in some logical notation). The implementations of the functions are given in a similar language.

> *Example*: CREATE_COMPONENT ($i_parent) creates a component and inserts it into the $i_parent component. $i_parent must exist, must be revisable (preconditions). A postconditions for $o_id (the created component id) is: $o_id is_contained_in $i_parent.

[4] PROGRAM GENERATION
The by now complete specification is programmed. RIDL-M generates automatically the basic operations. The whole specification together with the mapping is stored in a data dictionary to support query transformation from a high-level object-oriented query language to database level.

[5] USER MODEL
(could start from [2]). Defines semantics of user-interface (the general style, possible actions)
Example: a desktop metaphor is used for the document space. The user can create documents, move them over the screen, put them into a basket to destroy them etc.

[6] APPLICATION IMPLEMENTATION
The actions defined in the user model are decomposed to the level of the semantic operations defined in the conceptual model.

[7] APPLICATION SYNTAX DEFINITION
The actions defined in the user model are linked to keystrokes and syntactic patterns.

Our own experience draws primarily on the first four steps. The last three steps are just added for the sake of completeness. To be useful in this new methodology, the RIDL* tool needs some extensions. A declarative language to support stage [4] is needed, as well as a tool for generating and maintaining the data dictionary. Automatic program generation is an interesting research topic. We have made a beginning with the language definition in (Weigand, 1990).

Acknowledgements
The SPRITE project is sponsored by the European Community (ESPRIT) under grant 2001. More than twenty people at the participating companies and institutions are involved in this project. Their contributions are gratefully recognized.

References

DeTroyer, O., R. Meersman, F.Ponsaert, 1983. *RIDL User Guide,* Control Data DMRL Research Memorandum [available from the authors].

DeTroyer, O., R. Meersman & P.Verlinden, 1988. RIDL* on the CRIS Case: A Workbench for NIAM. In: Olle, T.W. et al (eds), *Computerized Assistance during the information systems life cycle*. Proc IFIP CRIS-88, North-Holland, Amsterdam.

DeTroyer, O., 1989. RIDL*: A tool for the computer-assisted engineering of large databases in the presence of integrity constraints. *Proc. ACM SIGMOD 89*, p418-429.

Intellibase, 1988. *RIDL* manuals*. Intellibase Inc, Antwerp, Belgium.

Moran, T.P., 1981. The command language grammar: a representation for the user interface of interactive computer systems. *Int. Journal Man-Machine Studies, 15,* p3-15.

Moran, T.P., 1983. Getting into a system: external-internal task mapping analysis. *Proc. CHI'83*, p45-49.

Nijssen, G.M., 1980. A framework for advanced mass storage applications. *Proc. MEDINFO 80, Tokyo*. North-Holland, Amsterdam.

Sybase, 1988. *Sybase Documentation.* Sybase Inc, Emeryville CA.

Verheijen, G.M.A. & J. van Bekkum, 1982. NIAM: An Information Analysis Method. In: T.W. Olle et al (eds) , *Information Systems Design Methodologies: a comparative review*. (CRIS-1), North-Holland, Amsterdam.

Weigand, H., 1990. An object-oriented approach in a multimedia databases project. *Accepted for IFIP TC2 Conf on Database Semantics (DS-4) Object-Oriented Dtabases, 2-6 July, at Windermere UK.*

OBJECT-ORIENTED INTEGRITY ENFORCEMENT IN A RELATIONAL ENVIRONMENT (*)

M.J. Briales, O. De Troyer
INFOLAB, EIT/Tilburg University
P.O. Box 90153, 5000 LE Tilburg, The Netherlands
fax: 31-13-663069, tel: 31-13-662688
briales@kub.nl, detroyer@kub.nl

ABSTRACT

When representing complex objects and their relationships in the context of a relational database management system, integrity constraint maintenance becomes an important issue because of the dependencies that are inherent to the relational representation of object-oriented structures. For the specification of integrity constraints, high-level, non-procedural mechanisms should be provided to the user, to insulate him from the low-level mechanisms that current RDBMSs provide for that purpose, if they provide any at all. We describe a schema-based approach to derive a set of integrity filters that support the storage and manipulation of complex objects on a RDBMS. The schema is defined with the NIAM conceptual data model. An extension to the current implementation of the RIDL* database design tool is proposed that will explicitly generate the set of integrity filters from the NIAM conceptual schema.

(*) This work is supported by the European Community under ESPRIT project 2001; SPRITE.

1. INTRODUCTION

An important topic of recent research work on database systems is the provision of adequate support for non-standard database applications such as multimedia databases, office automation, CAD/CAM databases etc.. Relational database systems (RDBMSs) fail to fulfil the requirements of these applications, which are characterized by complex data types and operations. The need of abstraction capabilities, such as support for hierarchical objects, shared sub-objects, dynamic object definitions, etc., is bringing an increasing interest for object-oriented concepts and techniques, and their integration with database technology. One approach is to extend the relational database system with user defined abstract data types and functions on these types. Examples of this approach are the POSTGRES data model (Rowe & Stonebraker 1987) and the extension to the SABRINA relational database management system (Gardarin et al. 1989). Other approaches such as EXODUS (Carey et al. 1986) provide the tools to build or generate database components from specifications. Several object-oriented database management systems can also be found in recent literature. GEMSTONE (Maier et al. 1986), ORION (Banerjee et al. 1987), and O_2 (Velez et al. 1989) are representative examples.

In the context of the European ESPRIT project, SPRITE (Storage, Processing and Retrieval in a Technical Environment), we have developed a MultiMedia Database (MMD) for the storage and maintenance of technical multimedia documentation (Dijkstra et al. 1989, Weigand 1990, Hederman & Weigand 1990). Because of their highly structured and multimedia contents, technical documents are complex entities, and as such difficult to model. Modelling techniques for complex objects were required. Although the feasibility of implementing the MMD on a suitable object-oriented DBMS is being evaluated (actually some research has been started on porting the MMD to the GEMSTONE database management system), issues as standardization and portability have determined our decision of choosing a RDBMS as storage system for the implementation of the MMD.

The approach to the MMD at this point in time is to build an object-oriented system on top of a classical relational database with well defined interfaces for complex object representation and manipulation. The interfaces are designed around integrity filters. An integrity filter is a set of explicit rules that a given object has to conform to. They are identified and specified during information analysis. All these rules, static as well as dynamic, can be seen as making up the semantics of our SPRITE system. SYBASE™ (SYBASE 1989) is the RDBMS that has been selected for the project because of its capability to support constraint checking and the procedural mechanisms that it provides, but our approach applies to any RDBMS with these properties.

In this paper we are concerned with the derivation of integrity filters in the context of the MMD database. A schema-based approach to derive a set of integrity filters for the objects identified in the application domain and the constraints defined on these objects, is presented. The schema is defined with the NIAM conceptual data model. Two classes of integrity filters are derived: 1) integrity filters that check the integrity constraints specified in the schema and forbid operations that do not satisfy these constraints, and 2) integrity filters that implement the elementary manipulation operations on the objects identified in the application domain, satisfying the set of integrity constraints specified for these objects. We propose to extend the current implementation of the RIDL* tool with the capability of explicitly generating both classes of constraint filters. RIDL* is a database engineering workbench based on NIAM. It allows the specification, verification and generation of database schemas (De Troyer et al. 1988, Intellibase 1988, De Troyer 1989).

The paper is organized as follows. In section 2 we give a set of preliminary definition and concepts concerning SYBASE™, the NIAM methodology and the RIDL* tool. Section 3 presents our schema-based approach to derive the set of integrity filters. Section 4 illustrates this approach with an example. In section 5 we conclude with a discussion.

2. PRELIMINARY DEFINITIONS AND CONCEPTS

2.1. Integrity Constraint Mechanisms Provided by SYBASE.

SYBASE™ (SYBASE 1989) provides two kinds of procedural mechanisms that can be used for implementing constraint specifications. These are *stored procedures* and *triggers*.

Stored procedures are collections of SQL statements, stored in the database as database objects. They are syntax checked and pre-compiled. The first time a stored procedure is executed, the data server query processor analyzes this procedure and stores an execution plan for it. Since most of the query-processing work has already been performed, subsequent execution of the stored procedure is fast. Stored procedures can take parameters and call other stored procedures. Default values can be assigned to parameters.

Triggers are a special kind of stored procedures that go into effect when a table is modified. Each trigger is specific to one or more of the data modification operations (update, insert or delete) and to a target table. For each trigger, the target table, the data modification command that will fire the trigger, and the trigger conditions and actions must be specified. Trigger conditions and actions are specified in terms of SQL statements. Trigger conditions specify additional criteria that determine whether the attempted insert, delete or update operation will cause the trigger action to be carried out. The trigger actions go into effect when the user action update, insert or delete is attempted. Each trigger can apply to only one table and a table can have a maximum of three triggers: one for insert, one for delete and one for update.

Both *stored procedures* and *triggers* are used in the MMD for the implementation of the integrity filters. *Triggers* implement integrity checks and *stored procedures* implement the elementary update operations on the MMD objects.

2.2. The NIAM Methodology and the RIDL* Tool

The RIDL* tool is a database engineering workbench (De Troyer et al. 1988, Intellibase 1988, De Troyer 1989) based on the NIAM (Nijssen Information Analysis Method) methodology (Nijssen 1976, Verheijen & Bekkum 1982).

NIAM (Nijssen 1976, Verheijen & Bekkum 1982) is a semantic network data model that uses the Binary Relationship Model. It is rich in constraint specifications and allows their graphical representation. NIAM makes a distinction between two kinds of object types: lexical and not lexical. *Lexical Object Types* (LOTs) are those concepts that can be expressed lexically, such as names, numbers, amounts, symbols, etc. *NOn-Lexical Object Types* have an abstract meaning, and Lexical Object Types can be considered as their lexical reference. All relationships between object types are expressed as *binary relationships*, called *fact types*. A *fact type* is made up of two *roles*, that express the roles that the two object types play in the relationship. Object types may be organized into subtypes using *sublink types*. They express an is-a relationship between two object types. The subtype occurrence implicitly inherits all properties of the supertype. In addition to these basic concepts, NIAM provides a notation for specifying a variety of constraints. Constraints constitute the semantical component of the conceptual schema. They express the knowledge of what is and is not allowed in the universe that constitutes the application domain.

RIDL* (De Troyer et al. 1988, Intellibase 1988, De Troyer 1989) assists the interactive design and development of the conceptual schema based on the graphical NIAM notation. From the specified conceptual schema, the tool generates relational schemas and additional constraint specifications for the semantics given in the conceptual schema. The relational schemas can be generated for any RDBMS. The generated relational schema together with the generated constraints is state equivalent with the given binary conceptual schema. The consistency of the set-algebraic constraints defined in the conceptual

schema on the population of roles and object types is first verified by the RIDL* tool.

RIDL* supports the graphical specification of the following constraints:

- *Identifier Constraint.* This constraint expresses the one-to-many restriction of a relationship (fact type). It represents functional dependency.
- *Fact Identifier Constraint.* The fact identifier constraint represents the many-to-many restriction of a relationship.
- *Total Role constraint.* This constraint states that each instance of an object type must participate in a given relationship.
- *Uniqueness Constraint.* The uniqueness constraint states that some combination of object instances identifies at most one other object instance.
- *Total Union Constraint.* This constraint is a generalization of a total role constraint, that states that each instance of the object type must participate in at least one of the indicated relationships.
- *Exclusion Constraint.* This constraint expresses the mutual exclusion of a number of subtypes.
- *Equality Constraint.* The equality constraint expresses the equal existence of a number of relationships for a given object instance.
- *Subset Constraint.* This constraint states that an instance can only participate in a certain relationship if it also participates in some other relationship.

For a detailed description on these constraints and an indication of their graphical notation we refer to De Troyer et al. (1988).

Our proposal is to extend the current implementation of the RIDL* tool with the capability of explicitly translating the integrity constraints specified in the conceptual schema to the integrity filters that we will describe in next section. At the moment of writing this article the implementation of the RIDL* tool is being extended with this feature. In its current available version, constraint types which have a corresponding constraint type in the target relational DBMS are generated and embedded in the database definition (De Troyer 1989). These are mainly functional dependencies, NOT-NULL conditions, check conditions and

for some systems referential integrity. Other constraints expressed in the conceptual schemas are only generated in an SQL-like fashion and added as comment lines. They may be encoded within applications in an ad-hoc manner. This is a consequence of the poorness of current RDBMSs in supporting constraints. Moreover, when constraints are violated, restoring the state of the database is limited in most RDBMSs to the traditional reversal action of undo or rollback of the current operation. This, however, does not always correspond with a logical operation.

In the extension that we present in this paper, the complete semantics expressed in the conceptual schema will be mapped to the procedural constraint mechanisms provided by SYBASE. Not only integrity checks and rollback actions are generated, but also integrity filters that propagate updates are derived. The last are defined as the elementary manipulation operations on the objects identified in the application domain. The closeness of NIAM to an object-oriented data model has permitted a very profitable use of the RIDL* tool in the design and implementation of the complex structures, identified in the application domain. From the conceptual schema, class definitions have been deduced. Primitive classes (data types) correspond to LOTs; NOLOTs represent abstract classes. Attributes of a class and class aggregations have been deduced from the fact types defined for a NOLOT. Sharing of subcomponents by complex objects is derived from fact identifier constraints. Generalization is represented in the sublink types.

3. SCHEMA-BASED DERIVATION OF INTEGRITY FILTERS

In this section an approach to automatically derive integrity filters from a binary conceptual schema is presented. Operations and constraints involved in the relational representation of object-oriented structures are analysed. Both structural information concerning the specification of object types and relationships in the conceptual schema and semantic information expressed by the integrity constraints on these object types and relationships are considered.

Two basic strategies are proposed for integrity maintenance. One is to forbid operations that do not satisfy integrity constraints. The second is to provide elementary manipulation operations that satisfy the integrity constraints for the object types identified in the application domain. In the latter, constraints are directly defined as the operations that preserve them. Update propagations are part of these operations.

3.1. Integrity Filters as Consistency Checks.

Based on the conceptual schema definition and the specified constraints, integrity filters defined as a set of conditions and actions, on the object types identified in the application domain, can be automatically derived. In this section we propose this derivation strategy.

The classes of constraints considered are fact identifier, total role, total union, exclusion, equality and subset, as they have been described in section 2.2. Other constraints mentioned in that section (role identifier, uniqueness constraints and total role which maps to check row) are already translated into corresponding RDBMS constraints by the RIDL* generator in its current implementation (De Troyer 1989).

Per object type in the conceptual schema, restricted by any of the constraint types mentioned above, we propose the automatic derivation of a restricted insert, a restricted update and a restricted delete filter that respectively disallow the operations insert, update and delete of the object, when a constraint is violated. Each integrity filter of this type specifies the considered object type, the operation, the integrity constraints associated to the filter, and the action that will be taken if any of these constraints is violated. Each integrity constraint associated to the filter is implemented as a condition that will be tested for false or true in all the objects involved in the constraint. When the condition becomes false, the action disallows the operation and activates a rollback mechanism that restores the state of the database.

The type of integrity filters defined above can be translated into SYBASE trigger mechanisms. The RIDL* generator is currently being extended to do so.

3.2. Integrity Filters as Predefined Operations on Complex Objects.

Based on the conceptual schema definition and the specified constraints, integrity filters that implement the elementary manipulation operations on the objects identified in the application domain, can be automatically derived. In this section we propose this derivation strategy.

In an object-oriented environment, elementary update operations on classes usually consist of instance creation, instance deletion, qualification and unqualification of an instance of a certain class and update of instance attributes. For set and list constructors the update operations insert and remove a member may also be considered. Per NOLOT in the conceptual schema we propose the automatic derivation of these operations as a set of integrity filters. As such, a consistent state of the database is guaranteed when these (and only these) operations are applied. Constraints are in this way directly defined as the operations that maintain them. Structural information concerning the specification of object types and relationships in the conceptual schema, as well as semantic information expressed by the integrity constraints on these object types and relationships, are incorporated in the inference mechanism that derives the operations.

Below, the generic specifications for the operations that we propose are presented. For each operation, the name, a short description, the object types of the conceptual schema the operation applies to, the generated input parameters, the preconditions and the postconditions are specified. The square brackets < and > are used to enclose non-literal symbols in a sentence. Conditions with the universal quantifier take the form FORALL A => B. Existential constraints start with EXIST or NOTEXIST. IF..THEN constructions are used to express actions that depend on conditions. Preconditions are considered both for generation and

for execution. They are conditions that must be satisfied before the operation is respectively generated or executed. Postconditions are the conditions that are true immediately after the operation has been executed. Structural and semantical information that is incorporated from the conceptual schema is specified in each operation. Constraints as the total role constraint described in section 2.2 and information derived from the sublink types, also described in that section, are particularly important because they specify the existence dependency relationships that characterizes object-oriented structures.

Flexibility is provided by giving the database designer the possibility to specify interactively in the generation process, a set of parameters. Designer supplied parameters are for instance the set of object types or combination of object types from the conceptual schema that a given operation should be generated for. By default, each operation is generated for every NOLOT or NOLOT combination of the schema. However, if the designer specifies a subset of them for a given operation, the operation is generated only for the objects in this subset. Attributes that are not restricted by a total constraint are optional. For these attributes, the database engineer can specify the subset to be included as input parameters in an operation. The aggregation to be considered for an operation may in this way be defined by the designer. For certain operations as forget and unqualify where uncontrolled change propagations could lead to the removal of many objects, due to the constraints, restricted and unrestricted versions may be derived. In the restricted versions, designer specified preconditions are incorporated, in such a way that the generated operation will only execute when the imposed preconditions hold.

The following operations have been defined:

Create_<OT_name>

Create an instance of the object type <OT_name>.

Per NOLOT specified in the conceptual schema, or for a user supplied subset of them, a create procedure is derived.

Generated Input Parameters:

- <primary_key>.

  Set of roles in the generated primary key for <OT_name>.

- <total_role_attributes>.

  Set of roles involved in total constraints and defined on <OT_name> or on any supertype of <OT_name>. During the generation process, the user can specify default values for these input parameters.

- <total_union_attributes>.

  Set of roles involved in total union constraints and defined on <OT_name> or on any supertype of <OT_name>. During the generation process, the user can specify default values for these input parameters.

- <optional_non_total_attributes>.

  Subset from set of roles not involved in total constraints, defined on <OT_name> or on any supertype of <OT_name>, and specified by the user during the generation process.

Preconditions at generation time:

- NOTEXIST total union for ANY subset of subtypes of <OT_name>.

Preconditions at execution time:

- NOTEXIST <primary_key> value for <OT_name>.
- NOTEXIST <primary_key> value for ANY supertype of <OT_name>.

Postconditions:

- EXIST <primary_key> value for <OT_name>.
- FORALL supertype of <OT_name> => EXIST a (possible other) primary key value that corresponds to <primary_key> value.
- ALL total and total-union constraints are satisfied for <OT_name>.
- FORALL supertype of <OT_name> => ALL total and total-union constraints are satisfied.
- FORALL parameter in <optional_non_total_attributes> => value is added OR value is set to NULL.

Qualify_< OT1_name>_as_<OT2_name>

Qualify an instance of the object type <OT1_name> as instance of its subtype <OT2_name>.

Per supertype-subtype combination (not necessarily direct subtype) specified in the conceptual schema, or for a user supplied subset of them, a qualify procedure is derived.

Generated Input Parameters:

- <primary_key1>.
  Set of roles in the generated primary key for <OT1_name>.
- <primary_key2>.
  Set of roles in the generated primary key for <OT2_name>.
- <total_role_attributes>.
  Set of roles involved in total constraints and defined on <OT2_name> or on any supertype of <OT2_name> that is subtype of <OT1_name>. During the generation process, the user can specify default values for these input parameters.
- <total_union_attributes>.
  Set of roles involved in total union constraints and defined on <OT2_name> or on any supertype of <OT2_name> that is subtype of <OT1_name>. During the generation process, the user can specify default values for these input parameters.
- <optional_non_total_attributes>.
  Subset from set of roles not involved in total constraints, defined on <OT2_name> or on any supertype of <OT2_name> that is subtype of <OT1_name>, and specified by the user during the generation process.

Preconditions at generation time:

- EXIST <OT1_name> supertype of <OT2_name>.
- NOTEXIST total union for ANY subset of subtypes of <OT2_name>.

Preconditions at execution time:

- EXIST <primary_key1> value for <OT1_name>.
- NOTEXIST <primary_key2> value for <OT2_name>.
- NOTEXIST <primary_key2> value for ANY supertype of <OT2_name> that is subtype of <OT1_name>.

Postconditions

- EXIST <primary_key2> value for <OT2_name>.
- EXIST correspondence between value of <primary_key2> and value of <primary_key1> for <OT1_name>.
- FORALL supertype of <OT2_name> that is subtype of <OT1_name> => EXIST correspondence between value of <primary_key2> and value of <primary_key1>.
- ALL total and total-union constraints are satisfied for <OT2_name>.
- FORALL supertype of <OT2_name> that is subtype of <OT1_name> => ALL total and total-union constraints are satisfied.
- FORALL parameter in <optional_non_total_attributes> => value is added OR value is set to NULL.

Forget_<OT_name>

Remove an instance of the object type <OT_name>.

Per NOLOT specified in the conceptual schema, or for a user supplied subset of them, a forget procedure is derived.

Generated Input Parameters:

- <primary_key>.
  Set of roles in the generated primary key for <OT_name>.

Preconditions at execution time:

- EXIST <primary_key> value for <OT_name>.

Postconditions

- NOTEXIST <primary_key> value for <OT_name>.

- NOT_EXIST primary key value that corresponds to <primary_key> value for ANY sub- or supertype of <OT_name>.
- NOTEXIST ANY relationship that involves the <OT_name> instance.

Forget_conditionally_<OT_name>

Remove an instance of the object type <OT_name> only if certain conditions are satisfied.

Per NOLOT specified in the conceptual schema, or for a user supplied subset of them, a conditional forget procedure is derived. During the generation process, the user must specify the conditions to be satisfied (e.g. <OT_name> instance not involved in a dependency relationship with another object type instance).

Generated Input Parameters:

- <primary_key>.
  Set of roles in the generated primary key for <OT_name>.

Preconditions at execution time:

- EXIST <primary_key> value for <OT_name>.

Postconditions

- IF conditions satisfied, THEN NOTEXIST <primary_key> value for <OT_name>.
- IF conditions satisfied, THEN NOTEXIST primary key value that corresponds to <primary_key> value for ANY sub- or supertype of <OT_name>.
- IF conditions satisfied, THEN NOTEXIST ANY relationship that involves the <OT_name> instance.

Unqualify_<OT2_name>_till_<OT1_name>

Unqualify an instance of the object type <OT1_name> as instance of its subtype <OT2_name> and as instance of all the intermediate object types.

Per supertype-subtype combination (not necessarily direct subtype) specified in

the conceptual schema, or for a subset of them supplied by the user, an unqualify procedure is derived.

Generated Input Parameters:

- <primary_key1>.
  Set of roles in the generated primary key for <OT1_name>.
- <primary_key2>.
  Set of roles in the generated primary key for <OT2_name>.

Preconditions at generation time:

- EXIST <OT1_name> supertype of <OT2_name>.
- NOTEXIST total union for ANY subset of subtypes of <OT1_name> that includes <OT2_name>.

Preconditions at execution time:

- EXIST <primary_key1> value for <OT1_name>.
- EXIST <primary_key2> value for <OT2_name>.

Postconditions

- NOTEXIST <primary_key2> value for <OT2_name>.
- NOTEXIST a correspondence between value of <primary_key2> and value of <primary_key1> in ANY supertype of <OT2_name> that is subtype of <OT1_name>.
- NOTEXIST ANY relationship that involves the <OT2_name> instance for <OT2_name>.
- NOTEXIST ANY relationship that involves the <OT2_name> instance for ANY supertype of <OT2_name> that is subtype of <OT1_name>.

Unqualify_conditionally_<OT2_name>_till_<OT1_name>

Unqualify an instance of the object type <OT1_name> as instance of its subtype <OT2_name> and as instance of all the intermediate object types, if certain conditions are satisfied.

Per supertype-subtype combination (not necessarily direct subtype) specified in the conceptual schema, or for a subset of them supplied by the user, a

conditional unqualify procedure is derived. During the generation process, the user must specify the conditions to be satisfied (e.g. <OT_name> instance not involved in a dependency relationship with another object type instance).

Generated Input Parameters:

- <primary_key1>.
  Set of roles in the generated primary key for <OT1_name>.
- <primary_key2>.
  Set of roles in the generated primary key for <OT2_name>.

Preconditions at generation time:

- EXIST <OT1_name> supertype of <OT2_name>.
- NOTEXIST total union for ANY subset of subtypes of <OT1_name> that includes <OT2_name>.

Preconditions at execution time:

- EXIST <primary_key1> value for <OT1_name>.
- EXIST <primary_key2> value for <OT2_name>.

Postconditions:

- IF conditions satisfied, THEN NOTEXIST <primary_key2> value for <OT2_name>.
- IF conditions satisfied, THEN NOTEXIST a correspondence between value of <primary_key2> and value of <primary_key1> for ANY supertype of <OT2_name> that is subtype of <OT1_name>.
- IF conditions satisfied, THEN NOTEXIST ANY relationship that involves the <OT2_name> instance for <OT2_name>.
- IF conditions satisfied, THEN NOTEXIST ANY relationship that involves the <OT2_name> instance for ANY supertype of <OT2_name> that is subtype of <OT1_name>.

Unqualify_till_<OT_name>

Unqualify an instance of the object type <OT_name> as instance of any of its subtypes.

Per object type specified in the conceptual schema, involved as supertype in a sublink type, or for a subset of them supplied by the user, an unqualify-till procedure is derived.

Generated Input Parameters:

- <primary_key>.
  Set of roles in the generated primary key for <OT_name>.

Preconditions at generation time:

- EXIST <OT_name> as supertype in a sublink type.
- NOTEXIST total union for ANY subset of subtypes of <OT_name>.

Preconditions at execution time:

- EXIST <primary_key> value for <OT_name>.

Postconditions

- NOTEXIST primary key value that corresponds to <primary_key> value for ANY subtype of <OT_name>.
- NOTEXIST relationship that involves the <OT_name> instance for ANY subtype of <OT_name>.

Update_<OT_name>

Set one or more attributes values of an instance of the object type <OT_name>. Per NOLOT specified in the conceptual schema, or for a subset of them supplied by the user, an update procedure is derived.

Generated Input Parameters:

- <primary_key>.
  Set of roles in the generated primary key for <OT_name>.
- <optional_attributes>.
  Subset from set of roles defined on <OT_name> or on any supertype of <OT_name> and specified by the user during the generation process.

Preconditions at execution time:

- EXIST <primary_key> value for <OT_name>.

Postconditions:

- FORALL parameter in <optional_attributes> => IF attribute value is specified THEN attribute value is updated.

Add_<OT2_name>_<role_name>_<OT1_name>

Add an instance of the fact type (relationship) where <role_name> is the role defined on <OT2_name> for this fact type.

Per fact type specified in the conceptual schema and restricted by a fact identifier constraint (many-to-many fact type) , or for a subset of them supplied by the user, an add procedure is derived.

Generated Input Parameters:

- <primary_key1>.
  Set of roles in the generated primary key for <OT1_name>.
- <primary_key2>.
  Set of roles in the generated primary key for <OT2_name>.

Preconditions at execution time:

- EXIST <primary_key1> value for <OT1_name>.
- EXIST <primary_key2> value for <OT2_name>.
- NOTEXIST <primary_key1>/primary_key2> combination value for the specified fact type.

Postconditions:

- EXIST <primary_key1>/<primary_key2> combination value for the given fact.

Remove_<OT2_name>_<role_name>_<OT1_name>

Remove an instance of the fact type (relationship) where <role_name> is the role defined on OT2_name for this fact type.

Per fact type specified in the conceptual schema and restricted by a fact identifier

constraint (many-to-many fact type), or for a subset of them specified by the user, a remove procedure is derived.

Generated Input Parameters:

- <primary_key1>.
  Set of roles in the generated primary key for <OT1_name>.
- <primary_key2>.
  Set of roles in the generated primary key for <OT2_name>.

Preconditions at execution time:

- EXIST <primary_key1>/<primary_key2> combination value for the given fact type.

Postconditions:

- NOTEXIST <primary_key1>/primary_key2> combination value for the given fact type.

For SYBASE, the generic operations described above can be implemented by means of stored procedures. The RIDL* generator is currently being extended to explicitly translate these operations.

4. EXAMPLE

In this section we present an illustrative example. A very small and simplified part of the MMD database of the SPRITE system is introduced in the example. The NIAM schema representing this part of the Universe of Discourse, as well as some of the integrity filters derived from this schema are presented.

Figures 1 and 2 show the NIAM schema. We concentrate in the example on the document, a central object of the SPRITE system. Figure 1 shows the type hierarchy around the document. In the root of the hierarchy the NOLOT doc_space_object is located. Document space objects are SPRITE objects that may appear in a user's document space. The NOLOTs folder and document are subtypes of the NOLOT doc_space_object. They have been grouped into a

subtype cluster. The subtypes implicitly inherit all properties of the supertype. The total union constraint for these subtypes (the ">" mark on the top part of the subtype cluster arrow) indicates that an instance of a doc_space_object is either a folder or a document. The exclusion constraint (the curved line connecting the different subtypes, and labeled by a small encircled symbol "X") expresses that the subtypes exclude each other. Subtypes of the object type document are also shown in the picture. They are also restricted by a total and an exclusion constraint. The NOLOT style_document is subtype of general_document. These two NOLOTS are also related by a fact type indicating that an instance of general_document may have a style that is given by a style_document. The notation for a fact type is a poligonal line connecting two object types and two contiguous boxes on it. The two boxes denote the roles played by these two object types in the relationship.

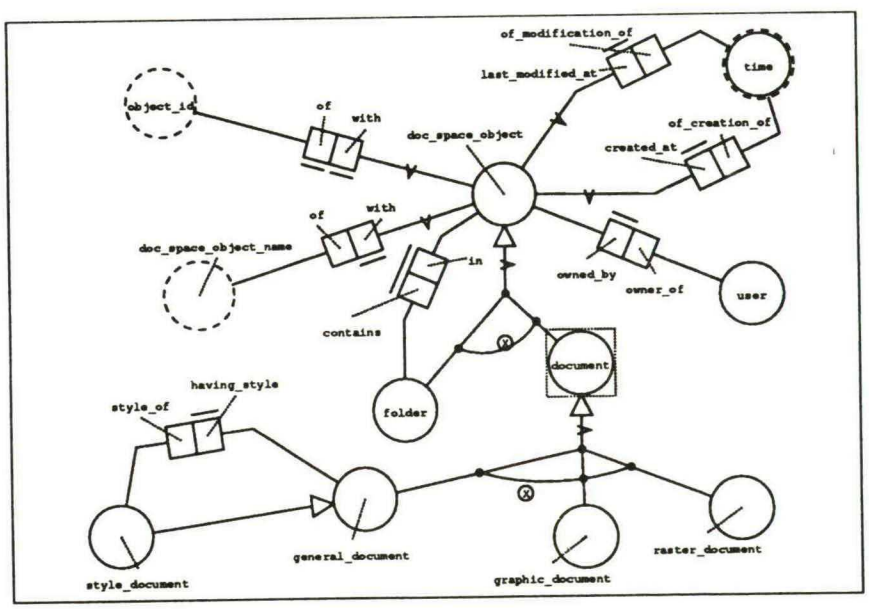

Figure1.Type hierarchy around the NOLOT document.

The relationships of the NOLOT doc_space_object with other object types by means of fact types are also shown in the picture. The total role constraints (the "V" markers) and the role identifier constraints (the bars above or under a role) indicate that every instance of doc_space_object is related to exactly one

doc_space_object_name, one time_of_creation, one time_of_modification and is owned by exactly one user. A doc_space_object may also be contained in a folder.

Figure 2 shows the NOLOT document and its relationships to other object types by means of fact types. The square including that NOLOT indicates that the NOLOT represents the same object in the two pictures of the schema.

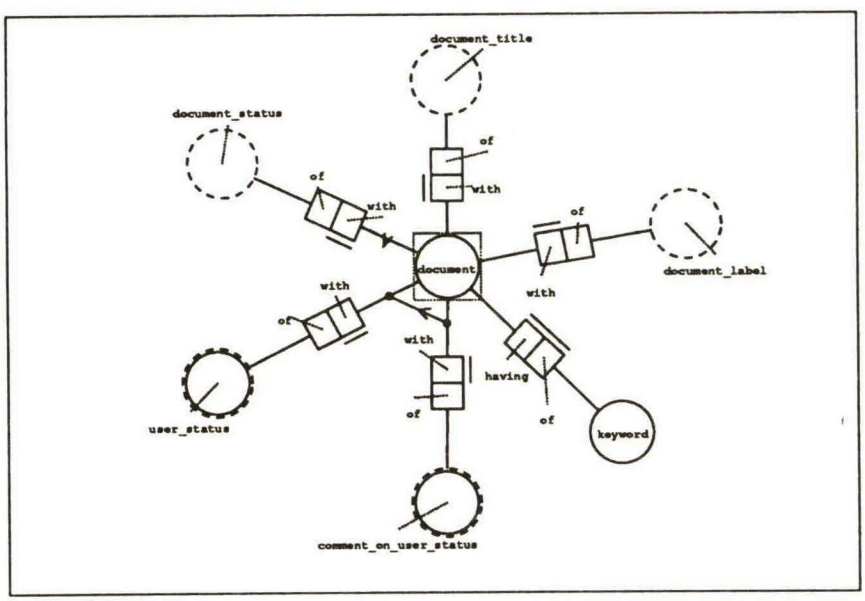

Figure 2. Document object type.

Every instance of document is related to zero or one document_title, zero or one document_label, zero or one user_status, zero or one comment_on_user_status and exactly one document_status. The subset constraint between user_status and comment_on_user_status indicates that a certain document can only participate in the role with_comment_on_user_status if it also participates in the role with_user_status. As indicated by the fact identifier constraint (the long bar above both roles), a document instance is related to zero or more keyword instances, and a keyword instance is related to zero or more document instances.

Below, the default mapping process that generates a fully normalized relational schema from the presented binary conceptual schema is briefly described. This

mapping process can be influenced by a number of mapping options that are available to the database engineer. As a result, more suitable and efficient data schemas than that provided by default can be generated for a particular application environment.

Per NOLOT (drawn as a solid named circle), RIDL* generates a relation (table) by grouping all functionally dependent roles for that NOLOT as attributes in one relation. For our example schema and the SYBASE data definition language, the table definition for document looks as follows:

```
CREATE TABLE document (
    object_id
        object_id
        NOT NULL,
    document_status_of
        document_status
        NOT NULL,
    document_title_of
        document_title
        NULL,
    document_label_of
        document_label
        NULL,
    user_status_of
        user_status
        NULL,
    comment_on_user_status_of
        comment_on_user_status
        NULL)
```

Per fact identifier constraint, a separate relation is created with only two attributes, one for each role of the fact. For our example schema the table document_keyword is constructed as follows:

```
CREATE TABLE document_keyword (
    object_id
        object_id
        NOT NULL,
    object_id_of
        object_id
        NOT NULL)
```

References to NOLOTs are replaced by one of their lexical representation types. In our example schema, the reference to the NOLOT document is replaced by its lexical representation object_id.

Constraint types from the presented NIAM schema, that map into a corresponding constraint type in the relational model, are identifier uniqueness

constraints, and total role constraints. Identifier uniqueness constraints map into SYBASE primary keys and unique indexes, and total constraints map into SYBASE NOT NULL constraints. For the document table the generated constraints look as:

```
sp_primarykey document, object_id
CREATE UNIQUE index C_KEY_22 ON document (object_id)
```

The keywords NULL and NOT NULL are generated for each attribute in the table according to the total constraints expressed in the conceptual schema (see table document above).

As described in section 3.1, constraints of the NIAM schema that do not have a corresponding constraint type in the relational schema map into SYBASE triggers. As an example, the SYBASE generated code for a restricted delete integrity filter is presented. When a delete statement on a document object is executed, SYBASE removes the row with the document object to be deleted from the document table and adds it to the temporal table "deleted". The integrity filter checks for the existence of the (primary key of the) deleted document object in other tables. In this case the tables that may contain document objects are the tables general_document, graphic_document and raster_document and the multivalued dependent table document_keyword. If the (primary key of the) object is encountered in any of these tables, the deletion is rejected and the transaction is rolled back in order to restore the state of the database. The trigger checks in this way constraints derived from the subtype hierarchy defined for document and prohibits dangling document objects.

```
CREATE TRIGGER FGT$document
    ON document
    FOR delete
AS
    IF EXISTS ( select *
                from general_document, deleted
                where general_document.object_id = deleted.object_id)
    BEGIN
        rollback transaction
        print "Deletion failed, subtype general_document exists"
    END
    ELSE IF EXISTS (    select *
                        from raster_document, deleted
                        where raster_document.object_id =
                        deleted.object_id)
    BEGIN
```

```
        rollback transaction
        print "Deletion failed, subtype raster_document exists"
    END
    ELSE IF EXISTS ( select *
                     from graphic_document, deleted
                     where graphic_document.object_id =
                     deleted.object_id)
    BEGIN
        rollback transaction
        print "Deletion failed, subtype graphic_document exists"
    END
    ELSE IF EXISTS ( select *
                     from document_keyword, deleted
                     where document_keyword.object_id =
                     deleted.object_id)
    BEGIN
        rollback transaction
        print "Deletion failed, document_keyword exists"
    END
GO
```

Predefined operations that preserve the integrity constraints are also generated from the NIAM schema. An example is the create operation for the NOLOT general_document. Optional and mandatory input parameters (as deduced from the total constraints in the conceptual schema) can be distinguished. They are respectively prefixed are by "i_" or "m_".

```
CREATE PROCEDURE CRF$general$document
@m_object_id object_id,        /* primary key */
/* total role attributes inherited from  doc_space_object */
@m_doc_space_object_name_of doc_space_object_name,
@m_time_of_creation_of time,
@m_time_of_modification_of time,
@m_name_owner_of name,
/* optional non-total  attributes inherited from doc_space_object */
@i_object_id_contains object_id = NULL,
/* total attributes inherited from document*/
@m_document_status_of document_status,
/* optional non-total attributes inherited from document */
/* Empty if no input values provided */
@i_document_title_of document_title = NULL,
@i_document_label_of document_label = NULL,
@i_user_status_of user_status = NULL,
@i_comment_on_user_status_of comment_on_user_status = NULL,
/* object type attributes */
@i_object_id_style_of object_id = NULL


AS
 /* Insert the primary key into the supertype table doc_space_object
  */
  insert into doc_space_object  (
    object_id,
    time_of_creation_of,
    time_of_modification_of,
    doc_space_object_name_of,
    name_owner_of)
  values  (
    @m_object_id,
```

```
    @m_time_of_creation_of,
    @m_time_of_modification_of,
    @m_doc_space_object_name_of,
    @m_name_owner_of)

/* Insert the primary key into the supertype table document
 */
 insert into document  (
    object_id,
    document_status_of,
    document_title_of,
    document_label_of,
    user_status_of,
    comment_on_user_status_of)
 values  (
    @m_object_id,
    @m_document_status_of,
    @i_document_title_of,
    @i_document_label_of,
    @i_user_status_of,
    @i_comment_on_user_status_of)

/* Insert the primary key into the table doc_space_object_folder:
 * Fact identifier constraint between doc_space_object and folder
 */
 if @i_object_id_contains != NULL
 begin
    insert into doc_space_object_folder  (
      object_id,
      object_id_contains)
    values  (
      @m_object_id,
      @i_object_id_contains)
 end

/* Insert the primary key into the object type table general_document
 */
 insert into general_document  (
    object_id,
    object_id_style_of)
 values  (
    @m_object_id,
    @i_object_id_style_of)

go
```

5. DISCUSSION

We have presented a schema-based approach to derive a set of integrity filters for a relational database. The schema is defined with the NIAM conceptual data model. Important in this approach is the possibility that it provides to identify the integrity constraints during the design phase, to specify the identified constraints in a high level non-procedural language, and to translate the high-level specified constraints into procedural integrity filters, that are embedded in the database definition and used at execution time for integrity maintenance.

We have considered two strategies for integrity constraint maintenance.

The first is to forbid operations that violate constraints. For this purpose we have defined integrity filters as integrity checks which are derived from the constraints specified in the conceptual schema. They include a rollback compensating action that restores the state of the database when constraints are violated. Approaches for efficiently checking constraints and efficient undo or rollback operations have also been proposed by a number of authors (Henschen et al. 1984, Hudson & King 1987, Qian & Smith 1987).

The second strategy that we have considered involves complex object manipulation operations that propagate changes in a database. We have defined for this purpose a set of integrity filters as the elementary manipulation operations on the object types identified in the application domain. They have been derived from structural and semantical information expressed in the conceptual schema. Similar approaches have recently been investigated by several researchers. Lindsay et al. (1987) consider a generic data management interface in a database management system architecture, designed to facilitate the implementation of data management extensions for RDBMSs, as part of the Starburst database project. They provide "generic operations" for relation modifications, and "attached procedures" for complex propagating operations that maintain the database integrity. Markowitz (1990) considers constraint enforcement in the context of relational schemas representing an Extension to the Entity-Relationship Model, and explores the capabilities of several RDBMSs to specify referential integrity constraints. Ceri & Widom (1990) provide a high level non-procedural language to specify constraints in relational databases, and derive from the specified constraints a set of production rules that maintain the constraints, by issuing actions to correct violations. Automatic derivation of compensating actions is reported as being still under research. Most papers mentioned above consider constraint enforcement, and automatic or semi-automatic execution of the data manipulation operations as the corrective actions whenever certain conditions are met. At run time the compensating actions must be chosen from the set of possible rules.

It is important to notice that when multiple rules are defined, several alternative actions may correct a given constraint violation. Finding an optimized enforcement strategy becomes a difficult task, due to the intrinsic complexity of general integrity constraints. The execution of one rule action may trigger a number of other rules. The order of firing operations can determine their effect. Although triggered propagation actions may give a high degree of flexibility to the system, the side effects that can be produced are dangerous and may cause other constraints to be violated. Predicting the state of the system becomes difficult under unexpected circumstances.

Several researchers have reported these problems and have dealt in different manner with them. Ceri & Widom (1990) use an algorithm to determine potential cyclic behaviour. The user, with the help of the system, should validate for each potential cyclic behaviour that termination in finite time is guaranteed. Casanova & Tucherman (1988) reduce the complexity of the system by restricting the set of integrity constraints to functional dependencies and referential integrity in a monitor that enforces them. The operations that the user submits in a session are either modified or propagated when these integrity constraints are not satisfied. Markowitz (1990) studies possible conflicts using the referential integrity graph that his model provides associated to the schema, and avoids rules that would produce conflicts. He also restricts the set of integrity constraints to functional dependencies and referential integrity. Another approach is to provide the system with correct database transactions rather than involving triggered propagation actions. Stemple et al. (1987) study several forms of feedback that can be generated from integrity constraints. They are provided to the designer of the database transaction to help him to find the appropriate transformation of an unsafe transaction into a safe transaction. Qian (1990) applies deductive program synthesis techniques to generate database transactions from logical specifications.

In the two strategies that we have presented, triggered propagation actions have been avoided. Only rollback mechanisms are automatically fired when an inconsistent state is reached. Change propagations in the database may only

occur by explicitly executing the elementary operations that have been defined for the system. The sequence of suboperations and the direction of changes in these operations are fixed, and therefore cyclic or unpredictable situations cannot occur. Consistent state of the system is guaranteed if changes to the database are performed through these operations. If other operations are used the checking filters will prohibit invalid changes. Flexibility is provided by the possibility that is given to the database designer, to specify interactively during the generation process a number of parameters. The aim of our approach has been to provide a safe operational object directed interface to the Sprite database that maintains the semantic integrity while offering reasonable run time performance. Since the principles of this interface are application independent, we have turned it into a general solution. Therefore, the current implementation of the RIDL* tool is being extended with the capability of automatically deriving such an interface.

In the Sprite System the checking filters are used for testing during the development phase. They constitute a valuable debugging tool. The generic operations provide a safe and efficient interface for the storage and manipulation of complex objects in the MMD database. They are the only valid interface that the MMD provides for update.

ACKNOWLEDGEMENTS

We thank Intellibase for valuable discussions.

REFERENCES

Banerjee, J., Chou, H., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H., 1987. Data Model Issues for Object-Oriented Applications. *ACM Trans. Office Information Systems*, 5(1), pp. 3-26.

Carey, M.J., De Witt, D.J., Frank, D., Graefe, G., Richardson J.E., Shekita E.J., Mauralikrishna, M., 1986. The Architecture of the EXODUS Extensible DBMS: A Preliminary Report. *Proc. Int. Workshop Object-Oriented Database Systems*, pp. 52-65.

Casanova, M.A., Tucherman, L., 1988. Enforcing Inclusion Dependencies and referential Integrity. *Proc. 14th VLDB Conf.* , pp. 38-49.

Ceri, S., Widom, J., 1990. Deriving Production Rules for Constraint Maintenance. *Proc. 16th VLDB Conf.*, pp. 566-577.

De Troyer, O., Meersman, R., Verlinden, P., 1988. RIDL* on the CRIS Case: A Workbench for NIAM. *Proc IFIP CRIS-88 Conf. Computerized Assistance during the Information System Life Cycle*. Olle, T.W. et al (eds.).

De Troyer, O., 1989. RIDL*: A Tool for the Computer-Assisted engineering of Large Databases in the Presence of Integrity Constraints. *Proc. ACM SIGMOD 89*, pp. 418-129.

Dijkstra, J., De Troyer, O., Meersman, R., Weigand, H., 1990. RIDL* as a Software Engineering AID - Some Practical Results. *Proc.4th Filin Conf. Methods and Tools as Aids to Design Information Systems*. (to appear).

Gardarin, G., Cheiney, J.P., Kiernan, G., Pastre, D., Stora, H. 1989. Managing Complex Objects in an Extensible Relational DBMS. *Proc. 16th VLDB Conf.*, pp. 55-65.

Hederman, L., Weigand, H., 1990. Versioned Objects in a Technical Documentation System. Proc. ESPRIT Conf. '90. (to appear).

Henschen, L.J., McCune, W.W., Naqvi, S.A., 1984. Compiling Constraint-Checking Programs from First-Order Formulas. *Advances in Database Theory*, 2, pp.145-169.Hudson, S.E., King, R., 1987. Object-Oriented Database Support for Software Environments. *Proc. ACM SIGMOD Conf.*, 16(3), pp 491-503.

Intellibase, 1988. *RIDL* manuals*. Intellibase Inc, Antwerp, Belgium.

Lindsay, B., McPherson, J., Pirahesh, H., 1987. A Data Management Extension Architecture. *Proc. ACM SIGMOD Conf.*, 16(3), pp. 220-226,

Maier, D., Stein, J., Otis, A., Purdy, A., 1986. Development of an Object-Oriented DBMS. *Proc. 1st OOPSLA Conf.*, pp. 472-484.

Markowitz, V.M., 1990. Referential Integrity Revisited: An Object-Oriented Perspective, *Proc. 16th VLDB Conf.*, pp 578-589.

Nijssen, G.M., 1976. A Gross Architecture for the Next Generation Database Management Systems. *Proc. IFIP TC-2 Conf. Modelling in Database Management Systems.* Nijssen, G.M. (ed.).

Qian, X., Smith, D.R., 1987. Integrity Constraint Reformulation for Efficient Validation. *Proc. 13th VLDB Conf.*, pp. 417-425.

Qian, X., 1990. Synthesizing Database Transactions. *Proc. 16th VLDB Conf.* pp. 552-565.

Rowe, L.A., Stonebraker, M.R., 1987. The POSTGRES Data Model. *Proc. 13th VLDB Conf.*, pp 83-96.

Stemple, D., Mazumdar, S., Sheard, T., 1987. On the Modes and Meaning of Feedback to Transaction Designers. *Proc. ACM SIGMD Conf.*, 16 (3), pp. 374-386.

Sybase, 1989. *SYBASE Documentation. Release 4.0.* Sybase Inc., Emeryville, California.

Velez, F., Bernard, G., Darnis, V., 1989. The O_2 Object Manager: an Overview. *Proc. 15th VLDB Conf.*, pp. 357-366.

Verheijen, G.M.A., van Bekkum, J., 1982. NIAM: An Information Analysis Method. *Proc. IFIP TC-8 Conf. Information Systems Design Methodologies: a Comparative Review (CRIS-1).* Olle, T.W. et al (eds.).

Weigand, H., 1990. An object-oriented approach in a multimedia database project. *IFIP TC-2 Conf. Database Semantics (DS-4) Object-Oriented Databases.* Kent,W., Meersman, R. (eds.). (to appear).

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS